Advanced Java Programming

Week 4 Topic Outline

Runtime Analysis

1. How do we know if our code is efficient?
2. Efficiency is measured in *time complexity* and in *space complexity*.

    - Usually we want to trade space for time - we have lots of space and little time.

3. Basic question: How many "steps" does our algorithm perform?

    - Need to decide what a "step" looks like for our algorithm. This should be the most basic repeated operation that takes a constant amount of time (not dependent on any variables).

4. Can't assume good conditions, so we analyze efficiency by assuming worst case scenario.
5. Other types of anlysis = "best case scenario" and "average case scenario"
6. Finding the upper bound O ("Big O Notation")

    - $O(1)$ Constant time
    - $O(n)$ Linear time
    - $O(n^2)$ Quadratic time
    - $O(n^3)$ Cubic time
    - $O(log_2 n)$ Logarithmic time
    - $O(n \cdot log_2 n)$ Linearithmic time
    - $O(n^k)$ Polynomial time
    - $O(2^n)$ Exponential time

7. If you want to learn more, take CPSC 365.
8. Practical rule: Only optimize when *necessary*.

    - Don't optimize until you already have something that works.
    - Do spend some time initially thinking through your algorithm (don't be obtuse)
    - Running programs typically spend 90% of the time in 10% of the code. Identify and optimize *that* part.

Abstract Data Types

An **abstract data type** specifies how we want to be able to access our data. It specifies an interface rather than an implementation.

Debatable point: Should complexity be part of the interface or the implementation?

Note: comments here on types apply to typed languages like Java, but not to untyped languages like Python or Ruby.

- Array
    - Fixed length
    - All terms have the same type
    - Random access (constant time)
    - Use: Bounded range of possibilities (counts, true/false, list with known length)
- List
    - Unbounded length
    - All terms have the same type
    - Sequential access
    - Use: Unbounded possibilities where any possibility can be accessed
- Record / Tuple
    - Fixed length
    - Each term is typed differently
    - We will discuss this one more later (time allowing)
    - Use: 2-3 values temporarily grouped together
- Stack
    - LIFO (last-in, first-out)
    - `push`, `pop`, `peek`
    - Unbounded length
    - All terms have the same type
    - Use: simulated card deck, elements to be processed in reverse order of insertion
- Queue
    - FIFO (first-in, first-out)
    - `add` / `enqueue`, `remove` / `dequeue`, `peek`
    - Unbounded length
    - All terms have the same type
    - Use: elements to be processed in same order as insertion
- Deque

    - Combination of stack & queue data types
    - Supports insertion and removal from both ends
    - Use: elements can be processed in either order and can be put back

- Priority Queue

    - `insert` inserts an element with a priority value
    - `pop` removes the element with the lowest value (which means highest priority)
    - `peek` looks at the element with the lowest value but does not modify
    - Unbounded length
    - All terms have the same type
    - Use: elements to be processed in order of priority

- Set

    - Collection that contains no duplicate elements
    - Often define: `union`, `intersection`, `difference`, `isSubset`, `isElementOf`
    - Unordered members
    - Unbounded length
    - All terms have the same type
    - Often random access (constant time)
    - Use: when we care about what we have seen, but not how many we have seen

- SortedSet

    - Collection that contains no duplicate elements
    - Often define: `union`, `intersection`, `difference`, `isSubset`, `contains`
    - Ordered members
    - Unbounded length
    - All terms have the same type
    - Often logarithmic access
    - Use: We need a set, but also want to be able to access by index, to sort, or to iterate sequentially.

- Map

    - Sometimes also called a "dictionary"
    - Collection of (key, value) pairs
    - All keys are unique
    - `put(k,v)` inserts a new (key, value) pair or reassigns a used key to a new value

- `get(k)` looks up the value associated with this key
- `remove(k)` removes the (key, value) pair with this key
- Keys have the same type, and values have the same type
- Use: Any time we want to store (key, value) pairs where we look up by key, any time we want to store a set but also associate a different value with each element in the set